Elaborazione di testi in *Mathematica*

Alternando gli esempi di argomento matematico con più accessibli problemi riguardanti la elaborazione non numerica, vediamo questo mese alcuni esempi di elaborazione di testi. E' doveroso notare che, poiché Mathematica è un linguaggio interpretato, i programmi che sperimenteremo saranno decine di volte più inefficienti dei loro colleghi scritti in C o in Pascal. Vale comunque la pena di prendere in considerazione l'implementazione in Mathematica tutte le volte che la compattezza e la semplicità di programmazione siano da ricercare anche a spese dell'efficienza.

di Francesco Romani

Calcolo di Anagrammi

Da quando mi sono preso l'impegno di scrivere un articolo al mese su *Mathematica* sono sempre alla ricerca di idee diverse, sia per mettere in evidenza i vari aspetti del sistema, sia per non perdere subito tutti i lettori sfornando una serie di articoli di matematica computazionale, tutti interessantissimi (per me) ma certamente di difficile lettura per i non matematici. Nel numero di Febbraio di MCmicrocomputer è apparso un interessante articolo di Corrado Giustozzi sugli algoritmi per il calcolo degli anagrammi. Ispirandoci a quel lavoro questo mese cerchiamo di ricostruire nel linguaggio di *Mathematica* alcuni dei suoi programmi. Iniziamo ricapitolando le idee base sul calcolo degli anagrammi per coloro che non possono procurarsi il numero di febbraio '93 con l'articolo originale dell'amico Corrado.
La **firma** o **anagramma principale** di una parola è l'anagramma in cui le varie lettere compaiono in ordine alfabetico, ognuna con la sua molteplicità. La funzione **Character** trasforma una stringa in una lista di caratteri e la funzione **Sort**, ordinando la lista ne genera la firma.

In[1]:=
```
Characters["pippo"]
```
Out[1]=
```
{p, i, p, p, o}
```
In[2]:=
```
Sort[Characters["pippo"]]
```
Out[2]=
```
{i, o, p, p, p}
```

Possiamo costruirci una funzione **firmato** che accoppia la firma alla parola per non perdere la preziosa informazione posizionale.

In[3]:=
```
firmato[x_]:={Sort[Characters[x]],x}
```
In[4]:=
```
firmato["pippo"]
```
Out[4]=
```
{{i, o, p, p, p}, pippo}
```

Con **Map** si applica **firmato** ad ogni elemento di una lista di stringhe; successivamente, ordinando la lista risultante si mettono vicine tutte le parole con la stessa stringa ovvero gli anagrammi.

In[5]:=
```
lista={"avrei", "ivrea", "perso", "varie",
    "preso"}
```
Out[5]=
```
{avrei, ivrea, perso, varie, preso}
```
In[6]:=
```
Map[firmato,lista]
```
Out[6]=
```
{{{a, e, i, r, v}, avrei}, {{a, e, i, r, v},
ivrea},
 {{e, o, p, r, s}, perso}, {{a, e, i, r, v},
varie},
 {{e, o, p, r, s}, preso}}
```
In[7]:=
```
Sort[%]
```
Out[7]=
```
{{{a, e, i, r, v}, avrei}, {{a, e, i, r, v},
ivrea},
 {{a, e, i, r, v}, varie}, {{e, o, p, r, s},
perso},
 {{e, o, p, r, s}, preso}}
```

Transpose permette di separare la due liste conservando l'ordine degli elementi in ciascuna lista.

In[8]:=
```
Transpose[%]
```
Out[8]=
```
{{{a, e, i, r, v}, {a, e, i, r, v}, {a, e,
i, r, v},
  {e, o, p, r, s}, {e, o, p, r, s}},
 {avrei, ivrea, varie, perso, preso}}
```

A questo punto costruiamo un programma completo che divide la lista stampando le sottoliste formate dalle parole che sono anagrammi le une delle altre. Il programma è un modulo con variabili locali **j,n,x,y.** In **x** viene messa la lista delle firme e in **y** la lista delle parole nell'ordine corrispondente. Si scandisce quindi con un **Do** la lista x fermandosi quando due elementi sono diversi e stampando una sottolista di almeno due parole con la stessa firma. Si noti che al posto della funzione esplicita **firmato** viene usata una funzione pura senza nome *(Vedi incorniciato).*

In[9]:=
```
anag[lista_]:=Module[ {j,n,x,y},
  n=Length[lista];
  {x,y}=Transpose[Sort[
```

```
Map[{Sort[Characters[#]],#}&,lista]];
  j=1;
  Do[If[(i==n)||(x[[i]]=!=x[[i+1]]),
        If[i>j, Print[Take[y,{j,i}]]];
        j=i+1],
  {i,n}]];
```

La lista delle firme viene scandita e si stampano le sottoliste di parole formate da 2 o più anagrammi.

In[10]:=

```
anag[lista]
```

```
{avrei, ivrea, varie}
{perso, preso}
```

Una semplice variazioni sul tema è un programma che stampa solo le sottoliste di più di k anagrammi, si noti che nonostante il nome della funzione sia sempre **anag** non c'è confusione perché la nuova definizione viene attivata solo se è chiamata con 2 argomenti.

In[11]:=

```
anag[lista_,k_]:=Module[ {j,n,x,y},
    j=1;
    n=Length[lista];
    {x,y}=Transpose[Sort[
       Map[{Sort[Characters[#]],#}&,
       lista]]];
    j=1;
    Do[If[(i==n)||(x[[i]]=!=x[[i+1]]),
     If[i>j+k-2,Print[Take[y,{j,i}]]];
       j=i+1],
    {i,n}]];
```

In[12]:=

```
anag[lista,3]
```

```
{avrei, ivrea, varie}
```

Per fare qualcosa di più corposo è necessario procurarsi una lista di parole italiane. Anch'io, come Giustozzi, sto mettendo insieme tutti i testi italiani che trovo per costruire un (rudimentale e incompleto) lessico di frequenza. Questo lessico, essendo costruito in modo dilettantesco e a partire da testi qualunque, contiene anche parole di non italiane e nomi propri. Allo scopo di cercare gli anagrammi delle parole più comuni ho preso la testa del file ed ho ottenuto un file di tipo TEXT di nome UU con circa mille tra le parole più frequenti. Per leggere un file tipo TEXT in *Mathematica* si può usare la funzione **Readlist** che restituisce una lista delle righe del file (nel nostro caso trattate come stringhe).

In[13]:=

```
lista=ReadList["UU",String];
```

In[14]:=

```
Length[lista]
```

Out[14]=

```
1433
```

Applicando **anag** si ottiene la lista degli anagrammi delle parole più comuni.

In[15]:=

```
anag[lista]
```

```
{AD, DA}
{AL, LA}
...
{ATTORNO, TORNATO}
{ORIGINE, REGIONI}
```

Con un file più grosso (circa 20000 parole), costruito nello stesso modo, si può fare un giochino diverso: cercare gli anagrammi di almeno tre parole analizzando via via porzioni più lunghe del file e calcolando il tempo occorrente. Questo ci permette da un lato di classificare gli anagrammi in base alla frequenza delle loro parole, dall'altro di valutare il costo del nostro algoritmo al variare della lunghezza delle liste a cui è applicato..

In[16]:=

```
lista=ReadList["ls11",String];
```

In[17]:=

```
Length[lista]
```

Out[17]=

```
20877
```

In[18]:=

```
Timing[anag[Take[lista,3000],3]]
```

```
{DEVO, DOVE, VEDO}
{AVREI, IVREA, VARIE}
{PERSO, PRESO, SPERO}
```

Out[18]=

```
{56.5333 Second, Null}
```

In[19]:=

```
Timing[anag[Take[lista,5000],3]]
```

```
{DEVO, DOVE, VEDO}
{AVREI, IVREA, VARIE}
...
{PERSO, PRESO, SPERO}
{ESTERA, SERATE, TERESA}
{APERTI, PARETI, PIETRA}
```

Out[19]=

```
{95.15 Second, Null}
```

In[20]:=

```
Timing[anag[Take[lista,10000],4]]
```

```
{ARMI, MARI, MIRA, RAMI}
{PERSA, PRESA, SAPER, SPERA}
{ATROCE, CREATO, CROATE, RECATO}
{DINAMO, DOMANI, DOMINA, NOMADI}
{AVERLI, RILEVA, RIVALE, RIVELA}
{PERITO, PIETRO, POTERI, POTREI, RIPETO}
```

Out[20]=

```
{191.55 Second, Null}
```

N.B.: Per l'elaborazione successiva servono almeno 15 Megabyte di memoria RAM!

In[21]:=

```
Timing[anag[lista,5]]
```

```
{ARMI, IRMA, MARI, MIRA, RAMI}
{ASPRE, PERSA, PRESA, SAPER, SPERA}
{ASPRO, PARSO, PROSA, SOPRA, SPARO}
{ATROCE, CREATO, CROATE, RECATO, TORACE}
{AVERLI, RILEVA, RIVALE, RIVELA, VIRALE}
{APERTI, PARETI, PATRIE, PIETRA, RIPETA}
{GARINO, GIRANO, IGNARO, IGNORA, INGRAO, OR-
GANI}
```

```
{PERITO, PIETRO, POTERI, POTREI, RIPETO}
{EVITARLO, RELATIVO, RILEVATO, RIVELATO,
VOLTAIRE}
```

Out[21]=

```
{411. Second, Null}
```

Un esperimento più grazioso e meno banale si può effettuare applicando **anag** ad una lista (ottenuta grazie alla cortesia degli autori del recente *Dizionario delle forme allterate della Lingua Italiana*) che contiene tutti gli alterati presenti nel dizionario stesso.

In[22]:=

```
lista=ReadList["ALT",String];
```

In[23]:=

```
Length[lista]
```

Out[23]=

```
13285
```

In meno di cinque minuti si ottengono tutti gli anagrammi di alterati che sono a loro volta alterati. La lista che segue mostra tutti i gruppi di almeno 3 anagrammi.

In[24]:=

```
Timing[anag[lista,3]]
```

```
{CANINO, NOCINA, ONCINA}
{OLIVINA, VIALINO, VIOLINA}
{OTTAVINA, OVATTINA, VOTATINA}
{ACERBINO, BRACIONE, BRIACONE}
{BARCHINO, BORCHINA, BRACHINO}
{ALBERINO, BARILONE, ERBOLINA}
{LUMACCIO, MALUCCIO, MULACCIO}
{ANNUCCIO, CUCINONA, NANUCCIO}
{PRESETTA, SERPETTA, SPERETTA}
{GELATINO, TEGLIONA, TEGOLINA}
{GRANETTO, RAGNETTO, ROGNETTA}
{OLIVETTA, VIALETTO, VIOLETTA}
{GRANACCIO, RAGNACCIO, ROGNACCIA}
{CASELLINO, COSELLINA, SCIALLONE}
{COPERTINA, PERTICONA, PRATICONE}
{CASETTINO, CASINETTO, COSETTINA, SCIATTONE}
{CATTIVONE, CIVETTONA, VOCETTINA}
{COLLARONE, CORALLONE, CORONELLA}
{AGRETTINO, ARGINETTO, REGINOTTA}
{NASTRETTO, STRANETTO, STRATTONE}
{CANTORUCCIO, CARTONUCCIO, CORNUTACCIO}
```

Out[24]=

```
{269.35 Second, Null}
```

Stringhe palindrome

Il secondo problema che prendiamo in considerazione è quella della selezione di stringhe palindrome, quelle parole cioè che possono essere lette sia da sinistra a destra che da destra a sinistra (per esempio ANNA). Usando la funzione **Characters**, già vista sopra, la **Reverse** (che inverte una lista) e il predicato **Equal** (che vale **True** se due espressioni sono uguali) è banale costruire un predicato **palin** che vale **True** se una stringa è palindroma.

In[25]:=

```
palin[x_]:=Characters[x]==Reverse[
  Characters[x]]
```

In[26]:=

```
palin["aia"]
```

Out[26]=

```
True
```

In[27]:=

```
palin["aida"]
```

Out[27]=

```
False
```

Applichiamo **palin** sull'insieme degli alterati e sulle 20000 parole del file ls11. La funzione **Select** seleziona solo gli elementi per cui **palin** vale **True** (ovvero le palindrome).

In[28]:=

```
lista=ReadList["ALT",String];
```

In[29]:=

```
Select[lista, palin]
```

Out[29]=

```
{ANIMINA}
```

L'unico alterato palindromo è ANIMINA. Proviamo adesso a cercare nelle 20000 parole del lessico di frequenza.

In[30]:=

```
lista=ReadList["ls11",String];
```

In[31]:=

```
Select[lista,palin]
```

Out[31]=

Pure funzioni

A volte capita di dovere definire una funzione che deve essere usata una volta sola. Ad esempio, per selezionare in una lista di coppie le coppie uguali tra loro, si potrebbe definite il predicato:

In[1]:=

```
Uguali[x_]==
x[[1]]==x[[2]];
```

e applicare **Select**:

In[2]:=

```
Select[{{a,b},
{c,c},{d,e}},
Uguali]
```

Out[3]=

```
{{c,c}}
```

E' possibile definire una funzione "pura" cioè priva di nome con una notazione simile a quella del lambda-calcolo

```
Function [{x},
x[[1]]==x[[2]]]
```

che ammette anche una rappresentazione compatta

```
(#[[1]]==#[[2]])&
```

dove # indica l'argomemto della funzione (se la funzione ha più argomenti questi sono indicati rispettivamente con #1, #2, ... ecc.).

La selezione di cui sopra diviene quindi

In[2]:=

```
Select[{{a,b},
{c,c},{d,e}},
(#[[1]]==#[[2]])&]
```

Out[3]=

```
{{c,c}}
```

Per finire vediamo alcuni esempi di funzioni in forma esplicita e nella forma pura (compatta). Gli ultimi esempi (nel riquadro qui sotto) sono usati anche nel testo.

```
f[x_]:= 2x                                  2#&
f[x_,y_]:= x+y                              (#1+#2)&
f[_]:= 2                                    2&
f[x_]:={Sort[Characters[x]],x}              {Sort[Characters[#]],#}&
f[x_]:=Count[l,x]                           Count[l,#]&
```

```
{E, I, NON, ALLA, O, AVEVA, OTTO, ORO, ANNA,
IRI, A, AMA, ALA, EBBE, IMI, ESSE, U, AEREA,
AIA, AFA, RADAR, POP, ADA, OSSO, ABEBA, IFI,
AGA, ELLE, AA}
```

Qui la scelta è più ricca, si notino le sigle di uso comune (IMI, IFI) e le parole non italiane (POP, ABEBA) che sono finite nella mia lista.

Calcolo della frequenza delle parole in un testo

Una delle elaborazioni più classiche che si possono effettuare su un testo è il calcolo della frequenza delle occorrenze delle varie forme, allo scopo di effettuare statistiche sulla lingua, sull'autore o su quello specifico testo. In un linguaggio di programmazione classico è necessario realizzare una opportune struttura dati (per esempio un albero binario di ricerca od una tabella hash) per memorizzare le varie parole con le loro occorrenze. Vedremo due implementazioni in *Mathematica* che fanno a meno di creare e gestire complicate strutture dati.
Cominciamo con definire una piccola lista di parole: il testo da esaminare.

In[32]:=
```
lista={sopra, la, panca, la, capra,
  campa,
      sotto, la, panca, la, capra,
  crepa};
```

L'unione delle lista la rende in ordine alfabetico senza ripetizioni

In[33]:=
```
parole=Union[lista]
```

Out[33]=
```
{campa, capra, crepa, la, panca, sopra, sot-
to}
```

La Funzione **Count** conta le occorrenze di una sottoespressione in un'espressione e quindi anche di una parola in una lista

In[34]:=
```
Count[lista,la]
```

Out[34]=
```
4
```

Creiamo una funzione che conta una generica parola in **lista** e applichiamola contando le occorrenze di tutte le parole

In[35]:=
```
conta[x_]:=Count[lista,x];
```

In[36]:=
```
Map[conta,parole]
```

Out[36]=
```
{1, 2, 1, 4, 2, 1, 1}
```

Appaiamo le occorrenze alle parole

In[37]:=
```
Sort[Transpose[{%,parole}]]
```

Out[37]=
```
{{1, campa}, {1, crepa}, {1, sopra}, {1,
  sotto}, {2, capra}, {2, panca}, {4, la}}
```

e invertiamo la lista per averla in ordine decrescente.

In[38]:=
```
Reverse[%]
```

Out[38]=
```
{{4, la}, {2, panca}, {2, capra}, {1, sot-
to}, {1, sopra}, {1, crepa}, {1, campa}}
```

Il tutto si mette insieme in una funzione **ContaFrequenze1**. Si noti la pura funzione **Count[l,#]&**.

In[39]:=
```
ContaFrequenze1[l_]:=Module[{parole},
  parole=Union[l];
  Reverse[Sort[Transpose[
        {Map[Count[l,#]&,
              parole],parole}]]]];
```

In[40]:=
```
ContaFrequenze1[lista]
```

Out[40]=
```
{{4, la}, {2, panca}, {2, capra}, {1, sot-
to}, {1, sopra}, {1, crepa}, {1,
  campa}}
```

La complessità di questo programma dipende sia dal numero n delle parole totali che dal numero m delle parole distinte (la cardinalità di Union[parole]). L'operazione più costosa è l'applicazione della funzione Count una volta per ogni parola distinta e il costo totale è circa proporzionale al prodotto n m. In questo articolo la trattazione del concetto di costo computazionale è abbastanza imprecisa per non appesantire la trattazione, vi prometto (minaccio?) una intera puntata dedicata all'argomento della complessità computazionale.
La seconda implementazione che presentiamo è simile al programma visto sopra per gli anagrammi: si ordinano le parole della lista da analizzare e si contano quelle vicine. L'operazione più costosa è l'ordinamento e il costo totale è circa proporzionale a n log n.

In[41]:=
```
ContaFrequenze2[lista_]:=Module[ {x,j,n,re-
sult},
  n=Length[lista];
  x=Sort[lista];
  result={};
  j=1;
  Do[If[(i==n)||(x[[i]]=!=x[[i+1]]),
    AppendTo[result,{i-j+1,x[[i]]}];
    j=i+1],
  {i,n}];
  Reverse[Sort[result]]];
```

In[42]:=
```
ContaFrequenze2[lista]
```

Out[42]=
```
{{4, la}, {2, panca}, {2, capra}, {1, sot-
to}, {1, sopra}, {1, crepa}, {1,
 campa}}
```

La terza implementazione è un po' particolare, e fa uso della tabella hash interna che è utilizzata nell'interprete di *Mathematica* per gestire le definizioni. Definiamo una lista vuota e una funzione che vale 0 per qualunque argomento.

In[43]:=
```
Clear[conta];
parole={};
conta[_]:=0;
```

Definiamo una funzione **add** che data una parola **x** effettua due azioni :
1) aggiunge **x** alla lista delle parole trovate;
2) definisce un valore particolare della funzione **conta** applicata a **x**: **conta[x]=conta[x]+1**. Ciò significa che se **x** è stata trovata per la prima volta **conta[x]** viene definito **1** altrimenti viene aumentato di una unità il valore precedentemente associato a **conta[x].**

In[44]:=
```
add[x_]:=(conta[x]=conta[x]+1;
          AppendTo[parole, x])
```

Applichiamo **add** a tutta la lista e vediamo come risulta definita **conta**

In[45]:=
```
Scan[add,lista]
```

In[46]:=
```
Definition[conta]
```

Out[46]=
```
conta[campa] = 1
conta[capra] = 2
conta[crepa] = 1
conta[la] = 4
conta[panca] = 2
conta[sopra] = 1
conta[sotto] = 1
conta[_] := 0
```

Di nuovo calcoliamo l'insieme delle parole trovate e applichiamo la funzione conta che ora ha una definizione tabellare.

In[47]:=
```
parole=Union[parole]
```

Out[47]=
```
{campa, capra, crepa, la, panca, sopra, sotto}
```

In[48]:=
```
Map[conta,parole]
```

Out[48]=
```
{1, 2, 1, 4, 2, 1, 1}
```

In[49]:=
```
Reverse[Sort[Transpose[{%,parole}]]]
```

Out[49]=
```
{{4, la}, {2, panca}, {2, capra}, {1, sotto}, {1, sopra}, {1, crepa}, {1,
 campa}}
```

Il tutto può essere messo in un unico programma. In questo caso le operazione più costose sono l'inserzione delle parole (costo proporzionale a n) e l'ordinamento delle coppie risultanti (costo m log m). Non è evidente se l'uso della tabella delle definizioni interna al linguaggio sia vantaggioso (la risposta l'avremo alla fine del capitolo).

In[50]:=
```
ContaFrequenze3[l_]:=Module[{add},
  Clear[conta];
  conta[_]:=0;
  add[x_]:=(conta[x]=conta[x]+1;
AppendTo[parole, x])
  parole={};
  Scan[add,l];
  parole=Union[parole];
  Reverse[Sort[Transpose[
        {Map[conta,parole],parole}]]]];
```

In[51]:=
```
ContaFrequenze3[lista]
```

Out[51]=
```
{{4, la}, {2, panca}, {2, capra}, {1, sotto}, {1, sopra}, {1, crepa},{1,
 campa}}
```

Vediamo ora di mettere alla prova i tre programmi con un testo più lungo: il testo dell'articolo introduttivo su Mathematica che ho scritto per il numero di Gennaio di questa rivista. Rimaneggiando un poco il testo si ottiene una lista di 1833 parole e simboli.

In[52]:=
```
lista=ReadList["MC1.txt",String];
Length[lista]
```

Out[52]=
```
1833
```

Applichiamo i tre programmi, siccome l'output è piuttosto lungo lo stampiamo nella forma **Short**, dove le parentesi angolate "<<", ">>" stanno a indicare il numero dei termini omessi.

In[53]:=
```
Timing[freq=ContaFrequenze1[lista];]
```

Out[53]=
```
{179.117 Second, Null}
```

In[54]:=
```
Short[freq]
```

Out[54]=
```
{{108, di},{51, e},{40, un},{29, la},
<<795>>, {1, ?},{1, +}}
```

In[55]:=
```
Timing[freq=ContaFrequenze2[lista];]
```

Out[55]=
```
{43.7 Second, Null}
```

In[56]:=
```
Timing[freq=ContaFrequenze3[lista];]
```

Out[56]=
```
{76.4667 Second, Null}
```

Si nota che la parola che ho usato più di frequente è "di" seguito da "e", "un" e "la". Si nota anche che il programma più artigianale è anche il più veloce e che l'uso della tabella delle definizioni interna al linguaggio è tutt'altro che vantaggioso.

MC

Bibliografia

C. Alberti, N.Ruimy, G.Turrini, G. Zanchi. La donzelletta vien dalla Donzella: Dizionario delle forme alterate della lingua italiana. Zanichelli, 1991.
C. Giustozzi. Anagrammatica 2: gli anagrammi multiparola. MCmicrocomputer n 126.
S. Wolfram. Mathematica. A System for Doing Mathematics by Computer. Addison Wesley,1991 (II Edition).

Francesco Romani è raggiungibile tramite Internet all'indirizzo romani@di.unipi.it